Python Lecture 1

Introduction to Python

1.1 Welcome

Python is a programming language. Like other languages, it gives us a way to communicate ideas. In the case of a programming language, these ideas are "commands" that people use to communicate with a computer!

We convey our commands to the computer by writing them in a text file using a programming language. These files are called *programs*. Running a program means telling a computer to read the text file, translate it to the set of operations that it understands, and perform those actions.

Example

```
my_name = "Jo"
print("Hello and welcome " + my_name + "!")
```

1.2 Comments

Ironically, the first thing we're going to do is show how to tell a computer to ignore a part of a program. Text written in a program but not run by the computer is called a *comment*. Python interprets anything after a `#` as a comment.

Comments can:

- Provide context for why something is written the way it is:

  ```
  # This variable will be used to count the number of times
  anyone tweets the word persnickety
  persnickety_count = 0
  ```

- Help other people reading the code understand it faster:

  ```
  # This code will calculate the likelihood that it will rain
  tomorrow
  complicated_rain_calculation_for_tomorrow()
  ```

- Ignore a line of code and see how a program will run without it:

  ```
  # useful_value = old_sloppy_code()
  useful_value = new_clean_code()
  ```

1.3 Print

Now what we're going to do is teach our computer to communicate. The gift of speech is valuable: a computer can answer many questions we have about "how" or "why" or "what" it is doing. In Python, the `print()` function is used to tell a computer to talk. The message to be printed should be surrounded by quotes:

```
# from Mary Shelley's Frankenstein
print("There is something at work in my soul, which I do not
understand.")
```

In the above example, we direct our program to `print()` an excerpt from a notable book. The printed words that appear as a result of the `print()` function are referred to as *output*. The output of this example program would be:

```
There is something at work in my soul, which I do not understand.
```

Example

```
print("Hello World!")
```

1.4 Strings

Computer programmers refer to blocks of text as *strings*. In our last exercise, we created the string "Hello world!". In Python a string is either surrounded by double quotes (`"Hello world"`) or single quotes (`'Hello world'`). It doesn't matter which kind you use, just be consistent.

Example

```
print('Jo')
```

1.5 Variables

Programming languages offer a method of storing data for reuse. If there is a greeting we want to present, a date we need to reuse, or a user ID we need to remember we can create a *variable* which can store a value. In Python, we *assign* variables by using the equals sign (`=`).

```
message_string = "Hello there"
# Prints "Hello there"
print(message_string)
```

In the above example, we store the message "Hello there" in a variable called `message_string`. Variables can't have spaces or symbols in their names other than an underscore (`_`). They can't begin with numbers but they can have numbers after the first letter (e.g., `cool_variable_5` is OK).

It's no coincidence we call these creatures "variables". If the context of a program changes, we can update a variable but perform the same logical process on it.

```
# Greeting
message_string = "Hello there"
print(message_string)
```

```
# Farewell
message_string = "Hasta la vista"
print(message_string)
```

Above, we create the variable `message_string`, assign a welcome message, and print the greeting. After we greet the user, we want to wish them goodbye. We then update `message_string` to a departure message and print that out.

Example

```
# We've defined the variable "meal" here to the name of the food we ate for br
eakfast!
meal = "Akara and bread"

# Printing out breakfast
print("Breakfast:")
print(meal)

# Now update meal to be lunch!
meal = "Eba and Egusi"

# Printing out lunch
print("Lunch:")
print(meal)

# Now update "meal" to be dinner
meal = "Rice and stew with chicken"

# Printing out dinner
print("Dinner:")
print(meal)
```

1.6 Errors

Humans are prone to making mistakes. Humans are also typically in charge of creating computer programs. To compensate, programming languages attempt to understand and explain mistakes made in their programs.

Python refers to these mistakes as *errors* and will point to the location where an error occurred with a `^` character. When programs throw errors that we didn't expect to encounter we call those errors *bugs*. Programmers call the process of updating the program so that it no longer produces unexpected errors *debugging*.

Two common errors that we encounter while writing Python are `SyntaxError` and `NameError`.

- `SyntaxError` means there is something wrong with the way your program is written — punctuation that does not belong, a command where it is not expected, or a missing parenthesis can all trigger a `SyntaxError`.
- A `NameError` occurs when the Python interpreter sees a word it does not recognize. Code that contains something that looks like a variable but was never defined will throw a `NameError`.

Example

```
print("This message has mismatched quote marks!')
print(Abracadabra)
```

1.7 Numbers

Computers can understand much more than just strings of text. Python has a few numeric *data types*. It has multiple ways of storing numbers. Which one you use depends on your intended purpose for the number you are saving.

An *integer*, or `int`, is a whole number. It has no decimal point and contains all counting numbers (1, 2, 3, …) as well as their negative counterparts and the number 0. If you were counting the number of people in a room, the number of jellybeans in a jar, or the number of keys on a keyboard you would likely use an integer.

A *floating-point number*, or a `float`, is a decimal number. It can be used to represent fractional quantities as well as precise measurements. If you were measuring the length of your bedroom wall, calculating the average test score of a seventh-grade class, or storing a baseball player's batting average for the 1998 season you would likely use a `float`.

Numbers can be assigned to variables or used literally in a program:

```
an_int = 2
a_float = 2.1

print(an_int + 3)
# Output: 5
```

Above we defined an integer and a float as the variables `an_int` and `a_float`. We printed out the sum of the variable `an_int` with the number `3`. We call the number 3 here a *literal*, meaning it's actually the number `3` and not a variable with the number 3 assigned to it.

Floating-point numbers can behave in some unexpected ways due to how computers store them. For more information on floating-point numbers and Python, review Python's documentation on floating-point limitations.

Example

```
# Define the release and runtime of any movie you know:
release_year = 2021
runtime = 102


# Define the rating out of 10:
rating_out_of_10 = 7.5
```

Calculations

Computers absolutely excel at performing calculations. The "compute" in their name comes from their historical association with providing answers to mathematical questions. Python performs addition, subtraction, multiplication, and division with `+`, `-`, `*`, and `/`.

```
# Prints "500"
print(573 - 74 + 1)

# Prints "50"
print(25 * 2)

# Prints "2.0"
print(10 / 5)
```

Notice that when we perform division, the result has a decimal place. This is because Python converts all `int`s to `float`s before performing division. In older versions of Python (2.7 and earlier) this conversion did not happen, and integer division would always round down to the nearest integer.

Division can throw its own special error: `ZeroDivisionError`. Python will raise this error when attempting to divide by 0.

Mathematical operations in Python follow the standard mathematical order of operations.

Example

```
print(25 * 68 + 13 / 28)
```

1.8 Changing Numbers

Variables that are assigned numeric values can be treated the same as the numbers themselves. Two variables can be added together, divided by `2`, and multiplied by a third variable without Python distinguishing between the variables and *literals* (like the number `2` in this example). Performing arithmetic on variables does not change the variable — you can only update a variable using the `=` sign.

```
coffee_price = 1.50
number_of_coffees = 4

# Prints "6.0"
print(coffee_price * number_of_coffees)
# Prints "1.5"
print(coffee_price)
# Prints "4"
print(number_of_coffees)

# Updating the price
coffee_price = 2.00

# Prints "8.0"
print(coffee_price * number_of_coffees)
# Prints "2.0"
print(coffee_price)
# Prints "4"
print(number_of_coffees)
```

We create two variables and assign numeric values to them. Then we perform a calculation on them. This doesn't update the variables! When we update the `coffee_price` variable and perform the calculations again, they use the updated values for the variable!

Example

```
price_of_pure_water = 10
plate_of_rice = 500
plantain = 100



price_of_pure_water = 20
plantain = 200
print(price_of_pure_water + plate_of_rice + plantain)
```

1.9 Exponents

Python can also perform exponentiation. In written math, you might see an exponent as a superscript number, but typing superscript numbers isn't always easy on modern keyboards. Since this operation is so related to multiplication, we use the notation `**`.

```
# 2 to the 10th power, or 1024
print(2 ** 10)

# 8 squared, or 64
print(8 ** 2)

# 9 * 9 * 9, 9 cubed, or 729
print(9 ** 3)

# We can even perform fractional exponents
# 4 to the half power, or 2
print(4 ** 0.5)
```

Here, we compute some simple exponents. We calculate 2 to the 10th power, 8 to the 2nd power, 9 to the 3rd power, and 4 to the 0.5th power.

1.10 Modulo

Python offers a companion to the division operator called the modulo operator. The modulo operator is indicated by `%` and gives the remainder of a division calculation. If the number is divisible, then the result of the modulo operator will be 0.

```
# Prints 4 because 29 / 5 is 5 with a remainder of 4
print(29 % 5)

# Prints 2 because 32 / 3 is 10 with a remainder of 2
print(32 % 3)

# Modulo by 2 returns 0 for even numbers and 1 for odd numbers
# Prints 0
print(44 % 2)
```

Here, we use the modulo operator to find the remainder of division operations. We see that `29 % 5` equals 4, `32 % 3` equals 2, and `44 % 2` equals 0.

The modulo operator is useful in programming when we want to perform an action every nth-time the code is run. Can the result of a modulo operation be larger than the divisor? Why or why not?

1.11 Concatenation

The `+` operator doesn't just add two numbers, it can also "add" two strings! The process of combining two strings is called *string concatenation*. Performing string concatenation creates a brand new string comprised of the first string's contents followed by the second string's contents (without any added space in-between).

```python
greeting_text = "Hey there!"
question_text = "How are you doing?"
full_text = greeting_text + question_text

# Prints "Hey there!How are you doing?"
print(full_text)
```

In this sample of code, we create two variables that hold strings and then concatenate them. But we notice that the result was missing a space between the two, let's add the space in-between using the same concatenation operator!

```python
full_text = greeting_text + " " + question_text

# Prints "Hey there! How are you doing?"
print(full_text)
```

Now the code prints the message we expected.

If you want to concatenate a string with a number you will need to make the number a string first, using the `str()` function. If you're trying to `print()` a numeric variable you can use commas to pass it as a different argument rather than converting it to a string.

```python
birthday_string = "I am "
age = 10
birthday_string_2 = " years old today!"

# Concatenating an integer with strings is possible if we turn the integer into a string first
full_birthday_string = birthday_string + str(age)
+ birthday_string_2

# Prints "I am 10 years old today!"
print(full_birthday_string)

# If we just want to print an integer
# we can pass a variable as an argument to
# print() regardless of whether
# it is a string.

# This also prints "I am 10 years old today!"
print(birthday_string, age, birthday_string_2)
```

Using `str()` we can convert variables that are not strings to strings and then concatenate them. But we don't need to convert a number to a string for it to be an argument to a print statement.

1.12 Plus Equals

Python offers a shorthand for updating variables. When you have a number saved in a variable and want to add to the current value of the variable, you can use the `+=` (plus-equals) operator.

```
# First we have a variable with a number saved
number_of_miles_hiked = 12

# Then we need to update that variable
# Let's say we hike another two miles today
number_of_miles_hiked += 2

# The new value is the old value
# Plus the number after the plus-equals
print(number_of_miles_hiked)
# Prints 14
```

Above, we keep a running count of the number of miles a person has gone hiking over time. Instead of recalculating from the start, we keep a grand total and update it when we've gone hiking further.

The plus-equals operator also can be used for string concatenation, like so:

```
hike_caption = "What an amazing time to walk through nature!"

# Almost forgot the hashtags!
hike_caption += " #nofilter"
hike_caption += " #blessed"
```

We create the social media caption for the photograph of nature we took on our hike, but then update the caption to include important social media tags we almost forgot.

Example

```
otal_price = 0


new_sneakers = 50.00


total_price += new_sneakers


nice_sweater = 39.00
fun_books = 20.00
# Update total_price here:
total_price += nice_sweater + fun_books
```

```
print("The total price is", total_price)
```

1.13 Multi-line Strings

Python strings are very flexible, but if we try to create a string that occupies multiple lines we find ourselves face-to-face with a `SyntaxError`. Python offers a solution: *multi-line* strings. By using three quote-marks (`"""` or `'''`) instead of one, we tell the program that the string doesn't end until the next triple-quote. This method is useful if the string being defined contains a lot of quotation marks and we want to be sure we don't close it prematurely.

```
leaves_of_grass = """
Poets to come! orators, singers, musicians to come!
Not to-day is to justify me and answer what I am for,
But you, a new brood, native, athletic, continental, greater than
  before known,
Arouse! for you must justify me.
"""
```

In the above example, we assign a famous poet's words to a variable. Even though the quote contains multiple linebreaks, the code works!

If a multi-line string isn't assigned a variable or used in an expression it is treated as a comment.

Example

```
to_you = """
Stranger, if you passing meet me and desire to speak to me, why
  should you not speak to me?
And why should I not speak to you?
"""


print(to_you)
```

1.14 User Input

How to assign variables with user input

So far, we've covered how to assign values directly in a Python file. However, we often want a user of a program to enter new information into program.

How can we do this? As it turns out, another way to assign a value to a variable is through user input.

While we output a variable’s value using print(), we assign information to a variable using input(). The input() function requires a prompt message, which it will print out for the user before they enter the new information.

Example

```
name = input("What is your name? ")
```

Lecture 1 Project

1. Block Letters

Use knowledge gained from class to produce the output displayed with the initials of your name

```
 SSS   L
S   S  L
S      L
 SSS   L
    S  L
S   S  L
 SSS   LLLLL
```

2. Receipts for Lovely Loveseats

We've decided to pursue the dream of small-business ownership and open up a furniture store called *Lovely Loveseats for Neat Suites on Fleet Street*. With our newfound knowledge of Python programming, we're going to build a system to help speed up the process of creating receipts for your customers.

In this project, we will be storing the names and prices of a furniture store's catalog in variables. You will then process the total price and item list of customers, printing them to the output terminal.

Adding In The Catalog

1. Let's add in our first item, the Lovely Loveseat that is the store's namesake. Create a variable called `lovely_loveseat_description` and assign to it the following string:

   ```
   Lovely Loveseat. Tufted polyester blend on wood. 32 inches
   high x 40 inches wide x 30 inches deep. Red or white.
   ```

2. Great, now let's create a price for the loveseat. Create a variable `lovely_loveseat_price` and set it equal to `254.00`.

3. Let's extend our inventory with another characteristic piece of furniture! Create a variable called `stylish_settee_description` and assign to it the following string:

```
Stylish Settee. Faux leather on birch. 29.50 inches high
x 54.75 inches wide x 28 inches deep. Black.
```

4. Now let's set the price for our Stylish Settee. Create a variable `stylish_settee_price` and assign it the value of `180.50`.
5. Fantastic, we just need one more item before we're ready for business. Create a new variable called `luxurious_lamp_description` and assign it the following:

```
Luxurious Lamp. Glass and iron. 36 inches tall. Brown with
cream shade.
```

6. Let's set the price for this item. Create a variable called `luxurious_lamp_price` and set it equal to `52.15`.
7. In order to be a business, we should also be calculating sales tax. Let's store that in a variable as well. Define the variable `sales_tax` and set it equal to `.088`. That's 8.8%.

Our First Customer

8. Our first customer is making their purchase! Let's keep a running tally of their expenses by defining a variable called `customer_one_total`. Since they haven't purchased anything yet, let's set that variable equal to `0` for now.

9. We should also keep a list of the descriptions of things they're purchasing. Create a variable called `customer_one_itemization` and set that equal to the empty string `""`. We'll tack on the descriptions to this as they make their purchases.

10. Our customer has decided they are going to purchase our Lovely Loveseat! Add the price to `customer_one_total`.

11. Let's start keeping track of the items our customer purchased. Add the description of the Lovely Loveseat to `customer_one_itemization`.

12. Our customer has also decided to purchase the Luxurious Lamp! Let's add the price to the customer's total.

13. Let's keep the itemization up-to-date and add the description of the Luxurious Lamp to our itemization.

14. They're ready to check out! Let's begin by calculating sales tax. Create a variable called `customer_one_tax` and set it equal to `customer_one_total` times `sales_tax`.

15. Add the sales tax to the customer's total cost.

16. Let's start printing up their receipt! Begin by printing out the heading for their itemization. Print the phrase `"Customer One Items:"`.

17. Print `customer_one_itemization`.

18. Now add a heading for their total cost: Print out `"Customer One Total:"`

19. Now print out their total! Our first customer now has a receipt for the things they purchased.

20. Congratulations! We created our catalog and served our first customer. We used our knowledge of strings and numbers to create and update variables. We were able to print out an itemized list and a total cost for our customer. Lovely!